CIS 431/531
Intro to Parallel Computing

CUDA

CUDA Program Structure

1) Declare memory on GPU to store the data to process
2) Copy data from host to device
3) Call the kernel(s) to process the data
4) Copy result back from device to host
5) Free memory

Example

Naïve

```
for(i = 0; i < N; i++)
{
    A[i] += 2;
}
```

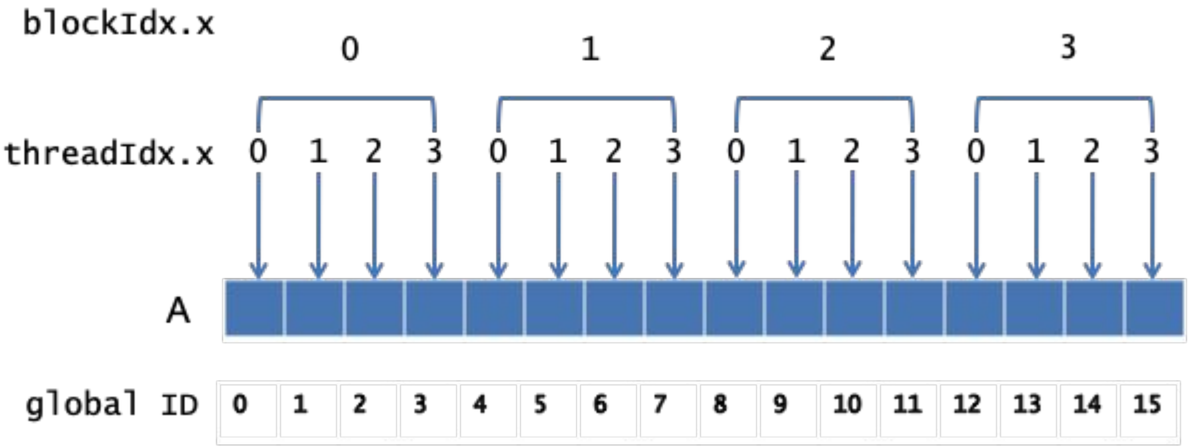
OpenMP

```
#pragma omp parallel for
for(i = 0; i < N; i++) {
    A[i] += 2;
}
```

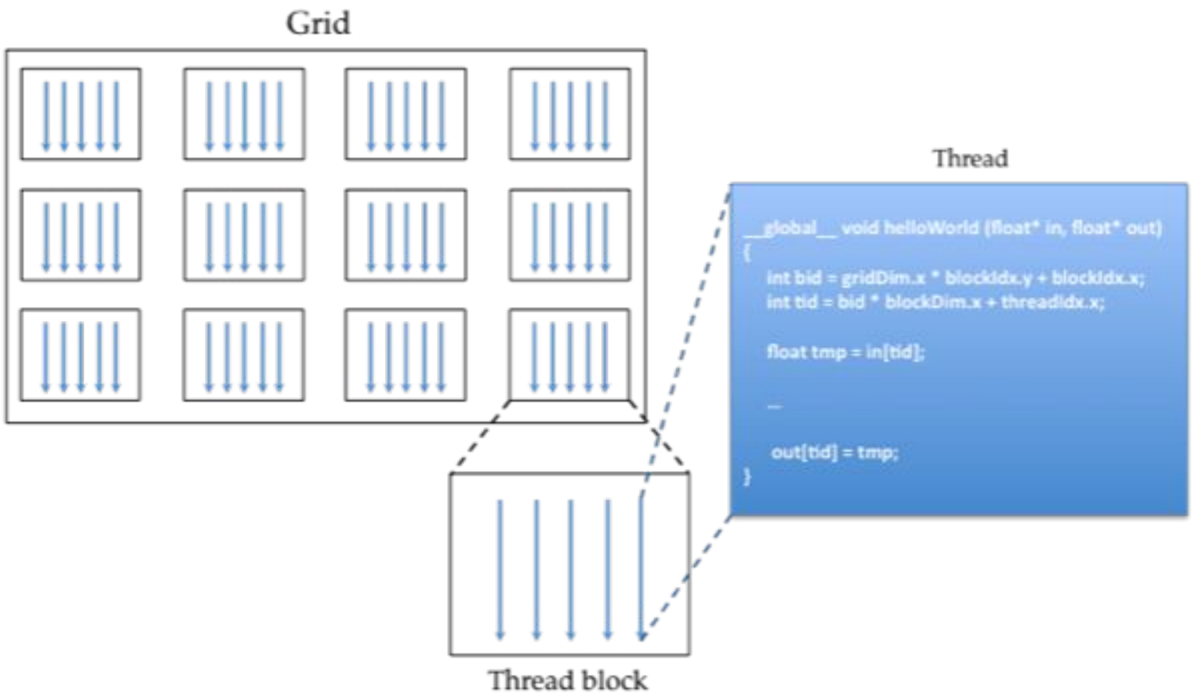
CUDA

```
int threadID = blockIdx.x * blockDim.x + threadIdx.x
A[threadID] += 2;
```

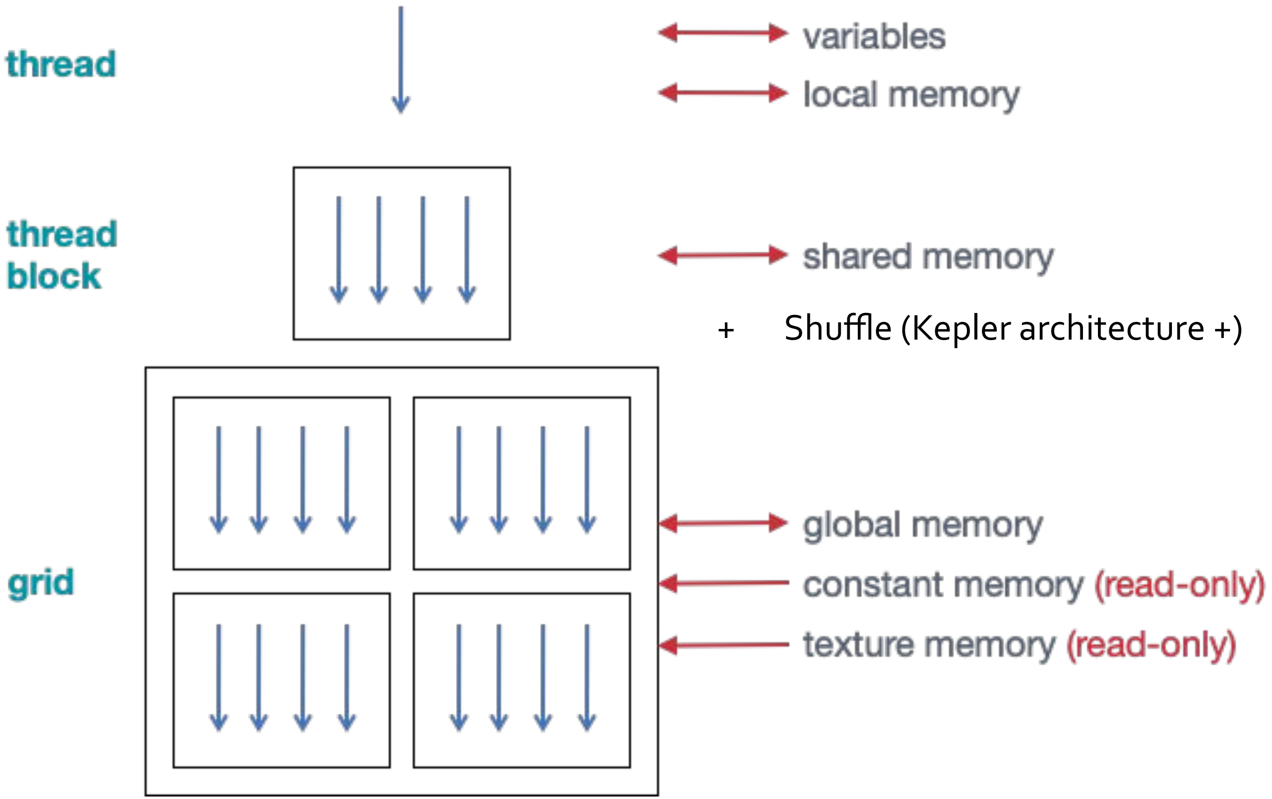
Mapping Threads



Thread Hierarchy



Memory Hierarchy



+ Shuffle (Kepler architecture +)

Performance Notes

Reading from the DRAM occurs at the granularity of 128 Byte transactions

- Requests are further decomposed to aligned cache lines
- L1 constant: 64 Bytes (Volta)
- L1 data: 32 Bytes (Volta)
- L2 cache: 64 Bytes (Volta)

Minimize loading redundant cache lines to maximize bandwidth utilization

- Aligned access to memory
- Sequential access pattern

Synchronization

Within a thread block

- via `__syncthreads();`

Global synchronization

- Implicit synchronization between **kernels**
- Only way to synchronize globally is to finish the **grid** and start another **grid**

Questions?

Shared Memory

Shared Memory

- Basically the same hardware as L1 cache
- Managed cache
- Can be configured as needed between L1 and shared memory (128 KB total)

Shared Memory

Let's use shared memory to do a matrix transpose

Matrix transpose

Shared Memory

Let’s use shared memory to do a matrix transpose

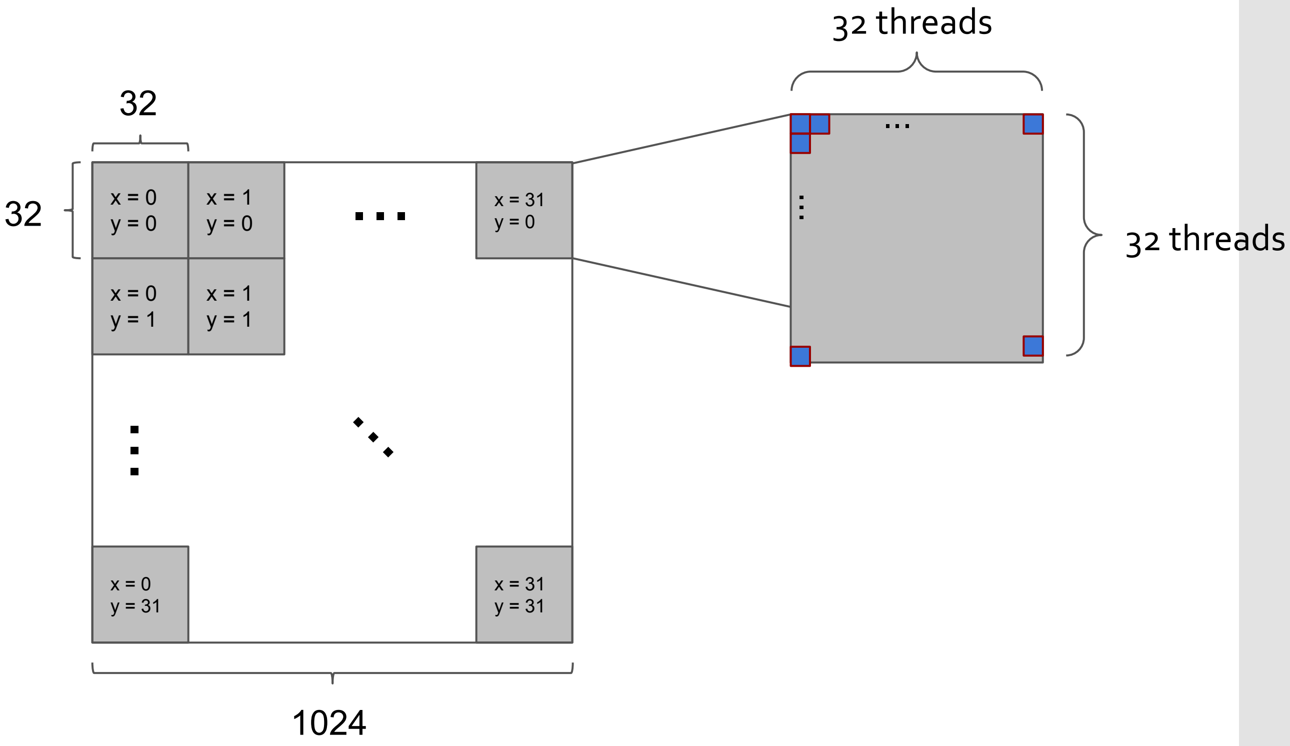
Matrix transpose

- Element in position (I, J) is moved to position (J, I)
- Alternatively, column I becomes row I

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}^T = \begin{bmatrix} a_{11} & a_{21} & a_{31} \\ a_{12} & a_{22} & a_{32} \\ a_{13} & a_{23} & a_{33} \end{bmatrix}$$

Shared Memory

One element to one thread mapping



Questions?

So far, we discussed having one thread handle one data element

This is not required - it’s often used because it’s simpler to have one thread handle one data element

However, it’s often better to give each thread more work to do, as creating threads and thread blocks does have an overhead cost (as long as you have enough thread to keep the GPU busy)

Shared Memory

Let's start with a naive code - what is the effective bandwidth of a simple memory to memory copy using threads?

Assume the matrix size is (`nx x ny`)

```
const int TILE_SIZE = 32; // we have to create smaller (thread) blocks to do the work
const int BLOCK_SIZE_X = TILE_SIZE;
const int BLOCK_SIZE_Y = 8; // each thread block is 32x8 (256) 2-D grid of threads

dim3 dimGrid(nx / TILE_SIZE, ny / TILE_SIZE, 1);// We are subdividing the matrix
into 32x32 blocks - how many elements in the matrix is each thread responsible for?

dim3 dimBlock(BLOCK_SIZE_X, BLOCK_SIZE_Y, 1);

copy<<<dimGrid, dimBlock>>>(d_cdata, d_idata);
```

Shared Memory

Let's start with a naive code - what is the effective bandwidth of a simple memory to memory copy using threads?

Assume the matrix size is (`nx x ny`)

```
const int TILE_SIZE = 32;
const int BLOCK_SIZE_X = TILE_SIZE;
const int BLOCK_SIZE_Y = 8;
dim3 dimGrid(nx / TILE_SIZE, ny / TILE_SIZE, 1);
dim3 dimBlock(BLOCK_SIZE_X, BLOCK_SIZE_Y, 1);
copy<<<dimGrid, dimBlock>>>(d_cdata, d_idata);
```

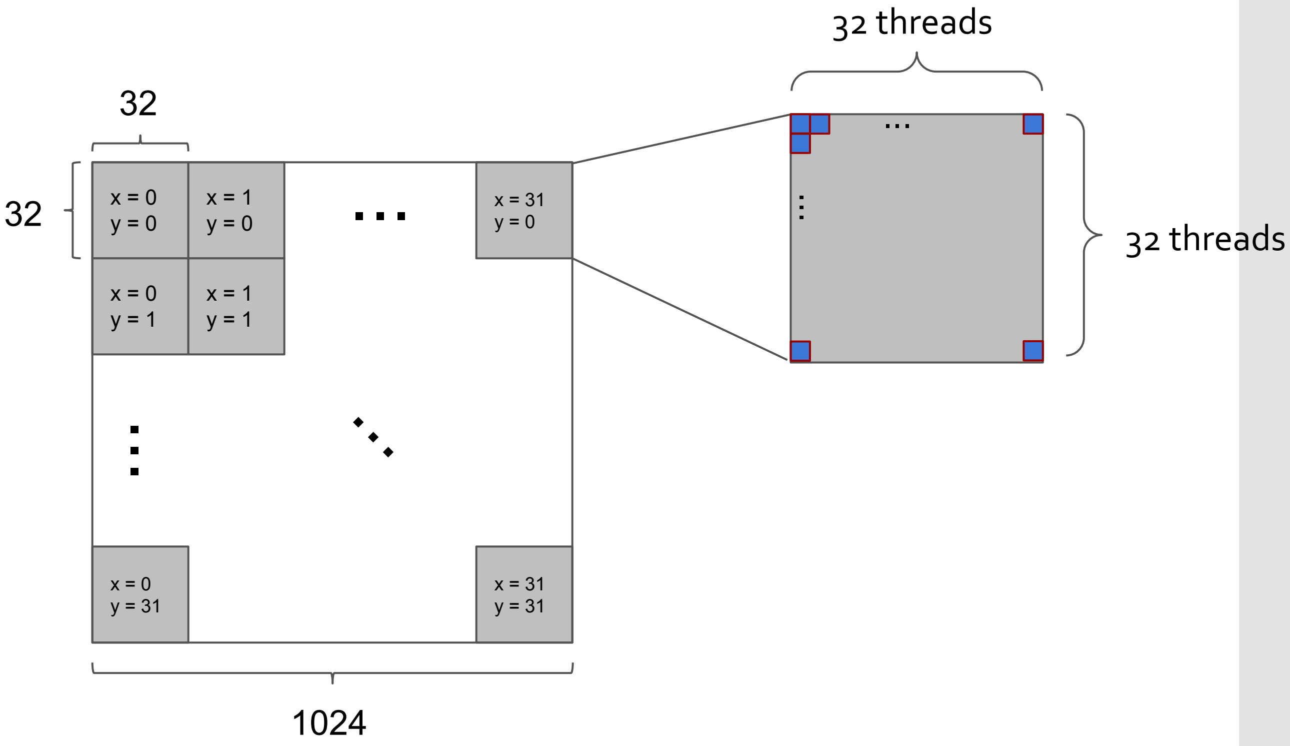
If the matrix 1024 x 1024 -> **Grid size** = 1024/32 x 1024/32 = **32 x 32** (2-D grid of thread blocks)

Thread block size is 32 x 8 -> each thread is responsible for (32 x 32) / (32 x 8) = 4 matrix elements

NOT a one (matrix element) to one (thread) mapping

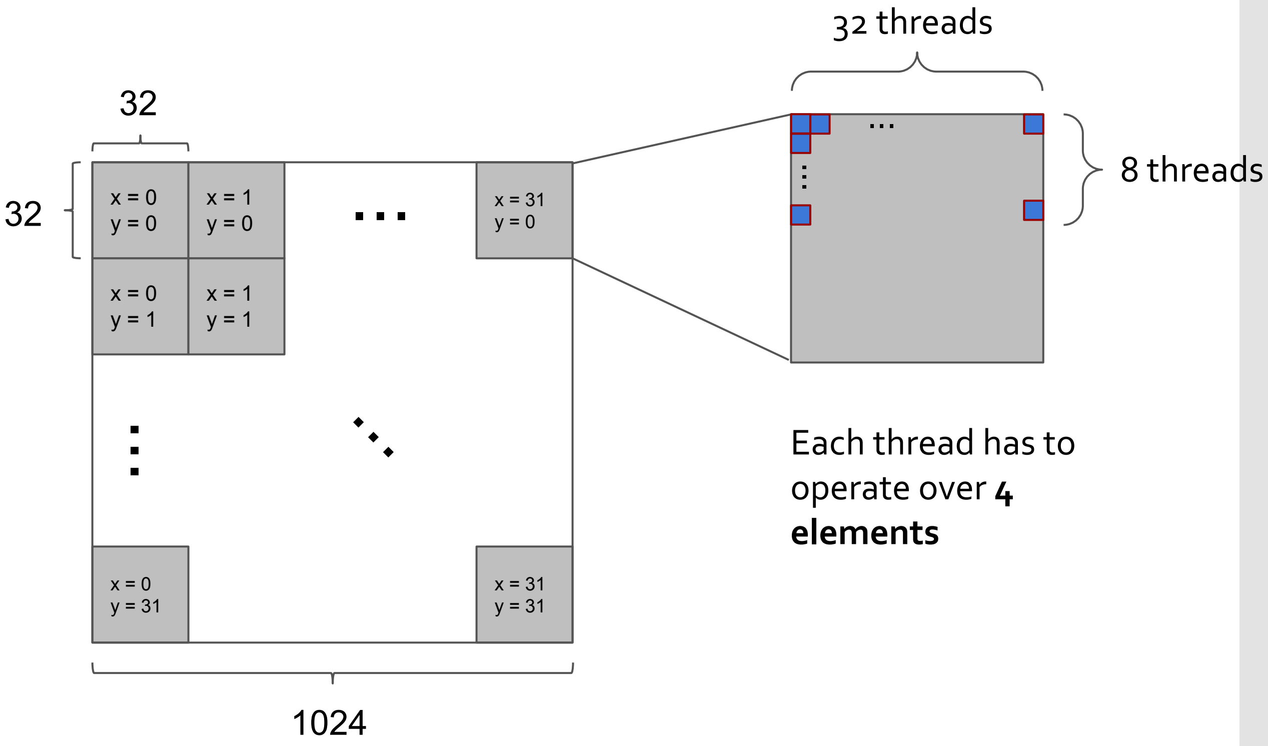
Shared Memory

One element to one thread mapping



Shared Memory

Many elements to one thread mapping



Each thread has to operate over **4 elements**

Mapping

Each thread block is responsible for 32x32 elements within the matrix

- The total number of thread blocks created is determined by what the thread blocks are responsible for

The thread blocks are made up of 32x8=256 threads

- The number of threads in the thread block DOES NOT have to be the total number of elements - we can have one thread be responsible for multiple elements

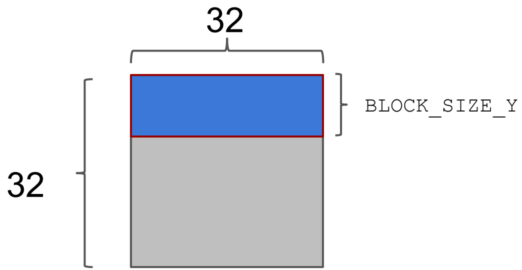
Shared Memory

Let’s start with a naive code - what is the effective bandwidth of a simple memory to memory copy using threads?

Assume the matrix size is (`ny x nx`)

```
const int TILE_SIZE = 32;
const int BLOCK_SIZE_X = TILE_SIZE;
const int BLOCK_SIZE_Y = 8;
dim3 dimGrid(nx / TILE_SIZE, ny / TILE_SIZE, 1);
dim3 dimBlock(BLOCK_SIZE_X, BLOCK_SIZE_Y, 1);
copy<<<dimGrid, dimBlock>>>(d_cdata, d_idata);

__global__ void copy(float *odata, const float *idata)
{
    int x = blockIdx.x * TILE_SIZE + threadIdx.x;
    int y = blockIdx.y * TILE_SIZE + threadIdx.y;
    int width = gridDim.x * TILE_SIZE;

    for (int j = 0; j < TILE_SIZE; j+= BLOCK_SIZE_Y) {
        odata[(y + j) * width + x] = idata[(y + j) * width + x];
    }
}
```

Determine which element (i,j) in the matrix each thread is responsible for (in the first iteration)



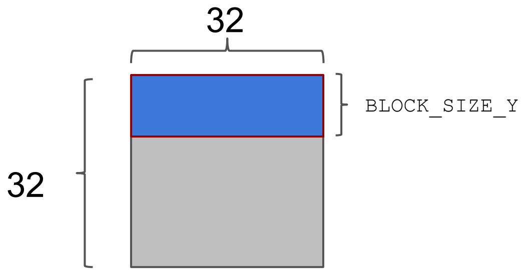
Shared Memory

Let's start with a naive code - what is the effective bandwidth of a simple memory to memory copy using threads?

Assume the matrix size is (`ny x nx`)

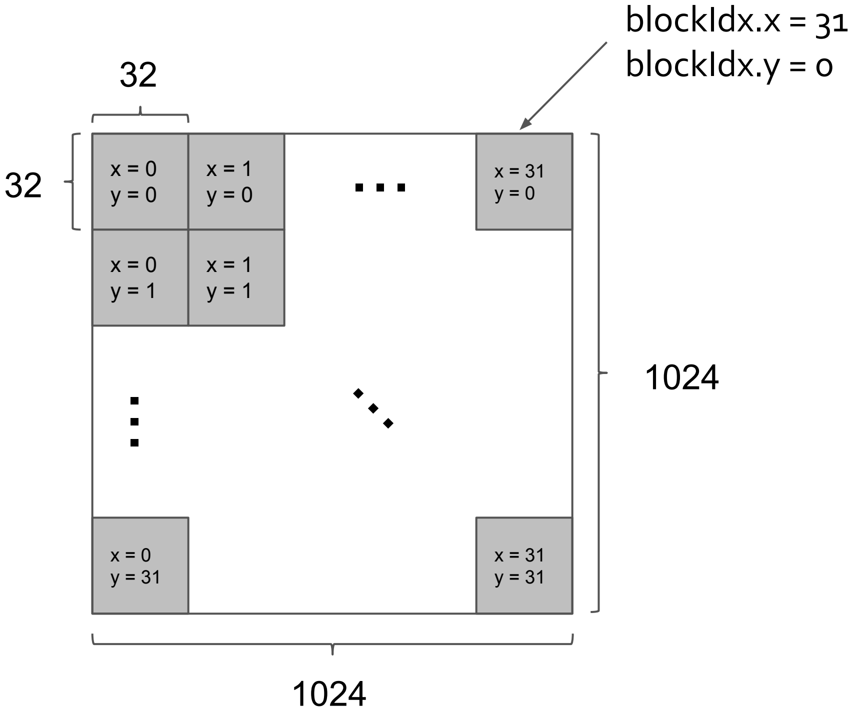
```
const int TILE_SIZE = 32;
const int BLOCK_SIZE_X = TILE_SIZE;
const int BLOCK_SIZE_Y = 8;
dim3 dimGrid(nx / TILE_SIZE, ny / TILE_SIZE, 1);
dim3 dimBlock(BLOCK_SIZE_X, BLOCK_SIZE_Y, 1);
copy<<<dimGrid, dimBlock>>>(d_cdata, d_idata);

__global__ void copy(float *odata, const float *idata)
{
    int x = blockIdx.x * TILE_SIZE + threadIdx.x;
    int y = blockIdx.y * TILE_SIZE + threadIdx.y;
    int width = gridDim.x * TILE_SIZE;

    for (int j = 0; j < TILE_SIZE; j+= BLOCK_SIZE_Y) {
        odata[(y + j) * width + x] = idata[(y + j) * width + x];
    }
}
```

Iterate over all the elements in the 32x32 tile in the matrix (4x in this example)



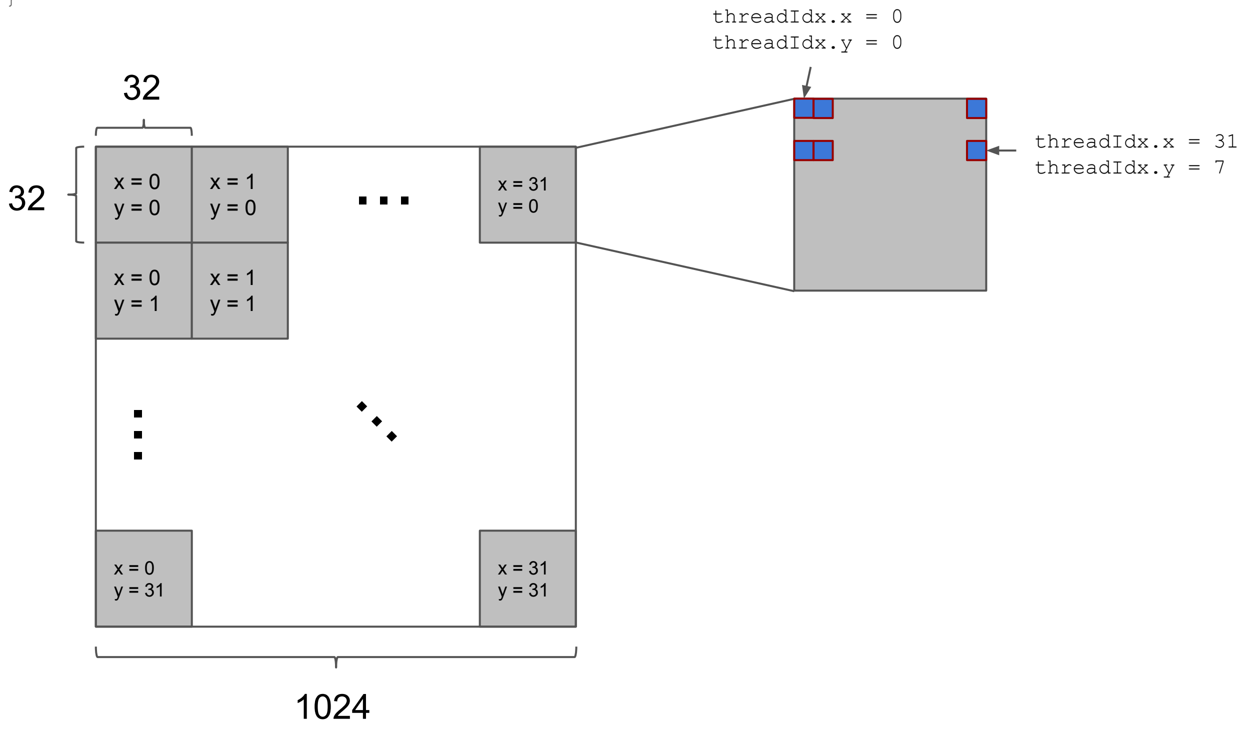
Shared Memory

```
__global__ void copy(float *odata, const float *idata)
{
    int x = blockIdx.x * TILE_SIZE + threadIdx.x;
    int y = blockIdx.y * TILE_SIZE + threadIdx.y;
    int width = gridDim.x * TILE_SIZE;

    for (int j = 0; j < TILE_SIZE; j+= BLOCK_SIZE_Y) {
        odata[(y + j) * width + x] = idata[(y + j) * width + x];
    }
}
```



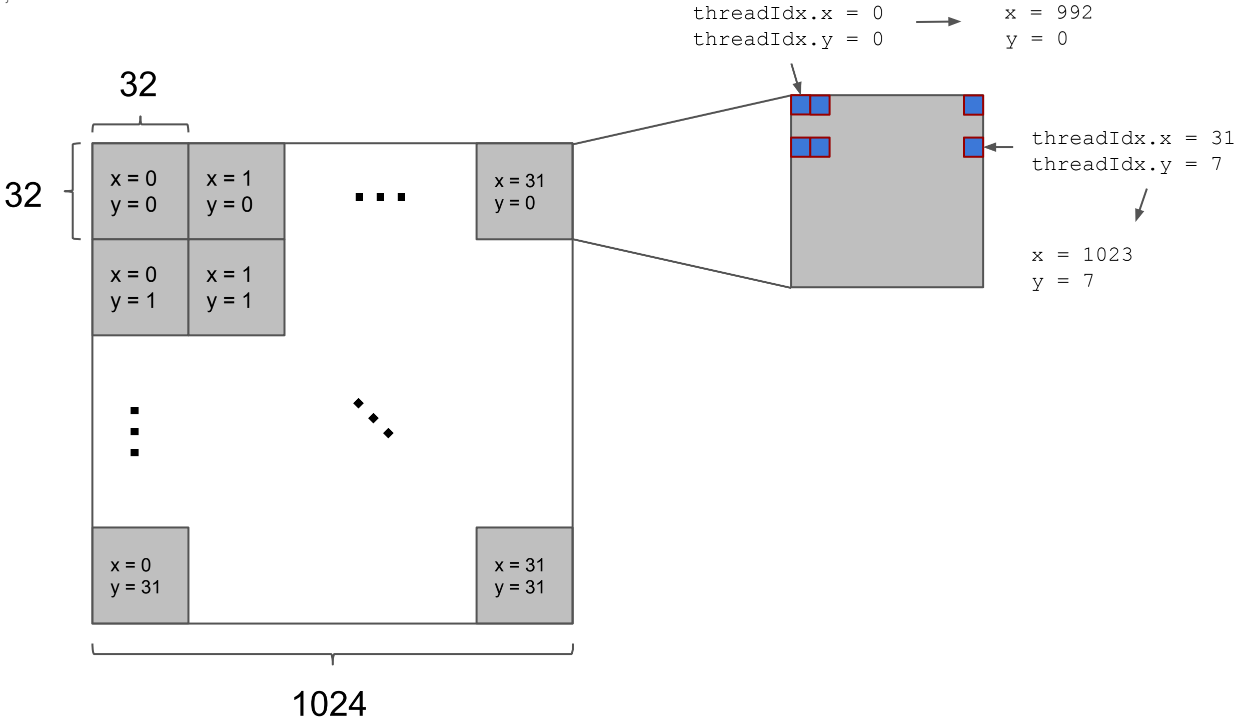
Shared Memory

```
__global__ void copy(float *odata, const float *idata)
{
    int x = blockIdx.x * TILE_SIZE  + threadIdx.x;
    int y = blockIdx.y * TILE_SIZE  + threadIdx.y;
    int width = gridDim.x * TILE_SIZE;

    for (int j = 0; j < TILE_SIZE; j+= BLOCK_SIZE_Y) {
        odata[(y + j) * width + x] = idata[(y + j) * width + x];
    }
}
```



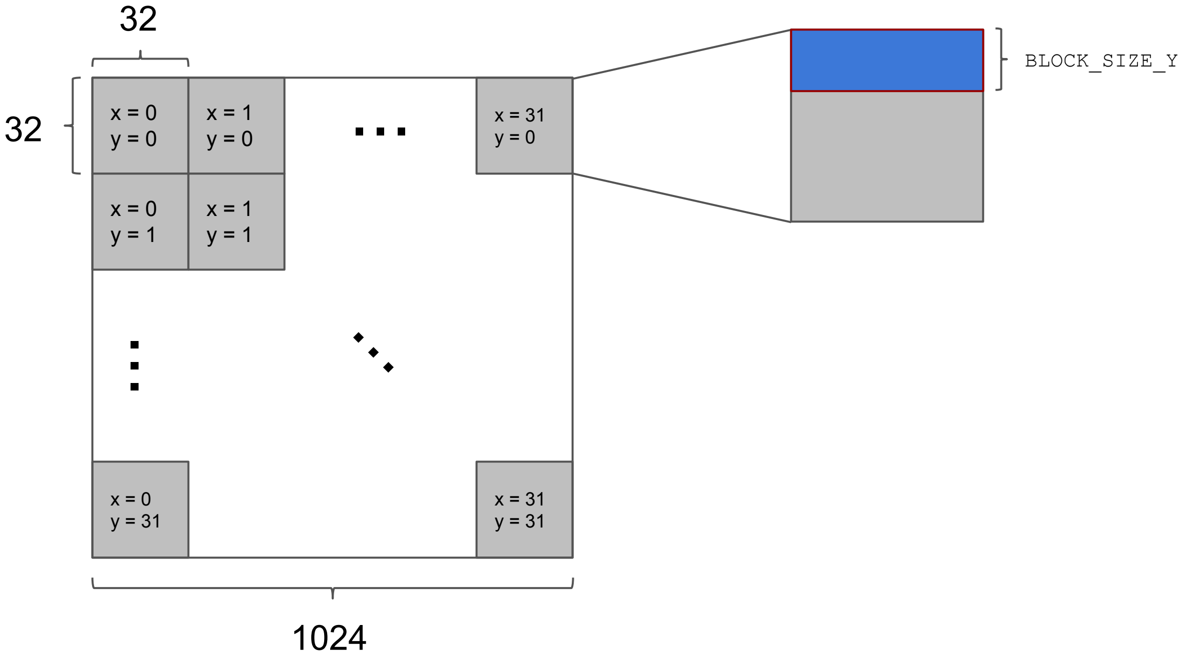
Shared Memory

```
__global__ void copy(float *odata, const float *idata)
{
    int x = blockIdx.x * TILE_SIZE + threadIdx.x;
    int y = blockIdx.y * TILE_SIZE + threadIdx.y;
    int width = gridDim.x * TILE_SIZE;

    for (int j = 0; j < TILE_SIZE; j+= BLOCK_SIZE_Y) {
        odata[(y + j) * width + x] = idata[(y + j) * width + x];
    }
}
```



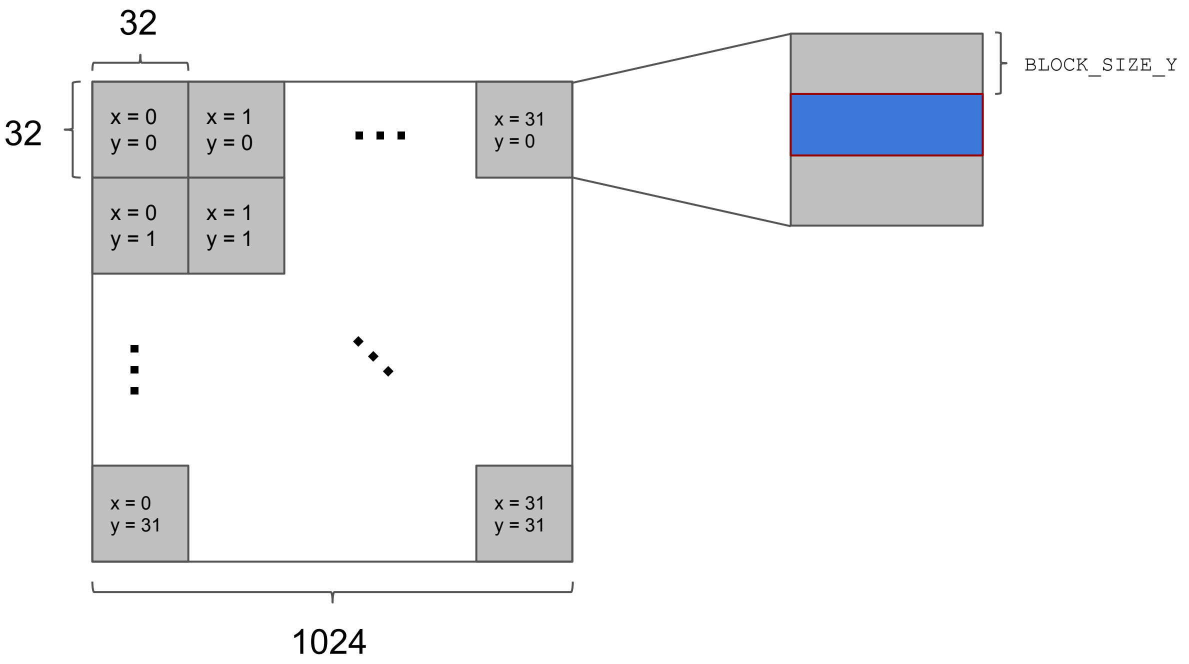
Shared Memory

```
__global__ void copy(float *odata, const float *idata)
{
    int x = blockIdx.x * TILE_SIZE + threadIdx.x;
    int y = blockIdx.y * TILE_SIZE + threadIdx.y;
    int width = gridDim.x * TILE_SIZE;

    for (int j = 0; j < TILE_SIZE; j+= BLOCK_SIZE_Y) {
        odata[(y + j) * width + x] = idata[(y + j) * width + x];
    }
}
```



Shared Memory

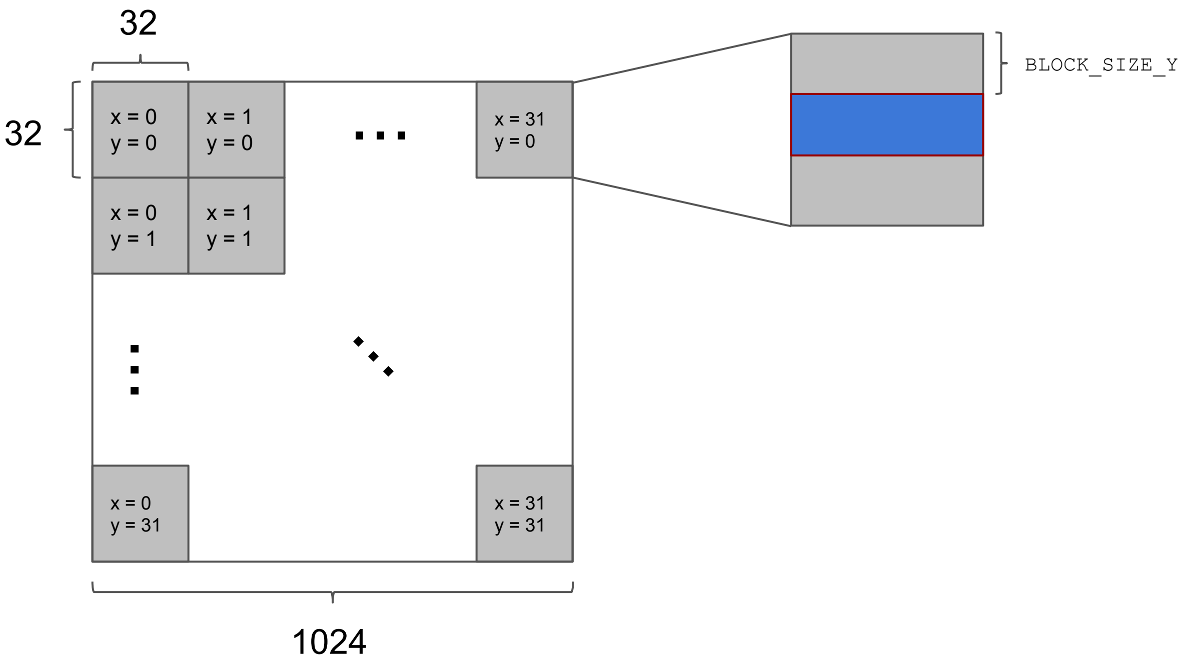
```
__global__ void copy(float *odata, const float *idata)
{
    int x = blockIdx.x * TILE_SIZE + threadIdx.x;
    int y = blockIdx.y * TILE_SIZE + threadIdx.y;
    int width = gridDim.x * TILE_SIZE;

    for (int j = 0; j < TILE_SIZE; j+= BLOCK_SIZE_Y) {
        odata[(y + j) * width + x] = idata[(y + j) * width + x];
    }
}
```



Shared Memory

```
__global__ void copy(float *odata, const float *idata)
{
    int x = blockIdx.x * TILE_SIZE + threadIdx.x;
    int y = blockIdx.y * TILE_SIZE + threadIdx.y;
    int width = gridDim.x * TILE_SIZE;

    for (int j = 0; j < TILE_SIZE; j+= BLOCK_SIZE_Y) {
        odata[(y + j) * width + x] = idata[(y + j) * width + x];
    }
}
```

Note that we are NOT doing a transpose (yet). We are doing a simple **copy** to see how long it takes (lower-bound on transpose time)



Shared Memory

Version	Bandwidth (GB/s)
copy	152.34

You can get about 170 GB/s on a simple memcpy.

Shared Memory

What happens if we use shared memory to do the same copy?

- Remember, we want to use shared memory to do the transpose

```
__global__ void copySharedMem(float *odata, const float *idata)
{
    __shared__ float tile[TILE_SIZE * TILE_SIZE];

    int x = blockIdx.x * TILE_SIZE + threadIdx.x;
    int y = blockIdx.y * TILE_SIZE + threadIdx.y;
    int width = gridDim.x * TILE_SIZE;

    for (int j = 0; j < TILE_SIZE; j += BLOCK_SIZE_Y) {
        tile[(threadIdx.y + j) * TILE_SIZE + threadIdx.x] =
            idata[(y + j) * width + x];
    }
    __syncthreads();

    for (int j = 0; j < TILE_SIZE; j += BLOCK_SIZE_Y) {
        odata[(y + j) * width + x] =
            tile[(threadIdx.y + j) * TILE_SIZE + threadIdx.x];
    }
}
```

Declare shared memory space

Shared Memory

What happens if we use shared memory to do the same copy?

- Remember, we want to use shared memory to do the transpose

```
__global__ void copySharedMem(float *odata, const float *idata)
{
    __shared__ float tile[TILE_SIZE * TILE_SIZE];

    int x = blockIdx.x * TILE_SIZE + threadIdx.x;
    int y = blockIdx.y * TILE_SIZE + threadIdx.y;
    int width = gridDim.x * TILE_SIZE;

    for (int j = 0; j < TILE_SIZE; j += BLOCK_SIZE_Y) {
        tile[(threadIdx.y + j) * TILE_SIZE + threadIdx.x] =
            idata[(y + j) * width + x];
    }
    __syncthreads();

    for (int j = 0; j < TILE_SIZE; j += BLOCK_SIZE_Y) {
        odata[(y + j) * width + x] =
            tile[(threadIdx.y + j) * TILE_SIZE + threadIdx.x];
    }
}
```

Each thread copies 4 elements to the shared memory

Shared Memory

What happens if we use shared memory to do the same copy?

- Remember, we want to use shared memory to do the transpose

```
__global__ void copySharedMem(float *odata, const float *idata)
{
    __shared__ float tile[TILE_SIZE * TILE_SIZE];

    int x = blockIdx.x * TILE_SIZE + threadIdx.x;
    int y = blockIdx.y * TILE_SIZE + threadIdx.y;
    int width = gridDim.x * TILE_SIZE;

    for (int j = 0; j < TILE_SIZE; j += BLOCK_SIZE_Y) {
        tile[(threadIdx.y + j) * TILE_SIZE + threadIdx.x] =
            idata[(y + j) * width + x];
    }
    __syncthreads();

    for (int j = 0; j < TILE_SIZE; j += BLOCK_SIZE_Y) {
        odata[(y + j) * width + x] =
            tile[(threadIdx.y + j) * TILE_SIZE + threadIdx.x];
    }
}
```

Each thread copies 4 elements from shared memory to odata[]

Shared Memory

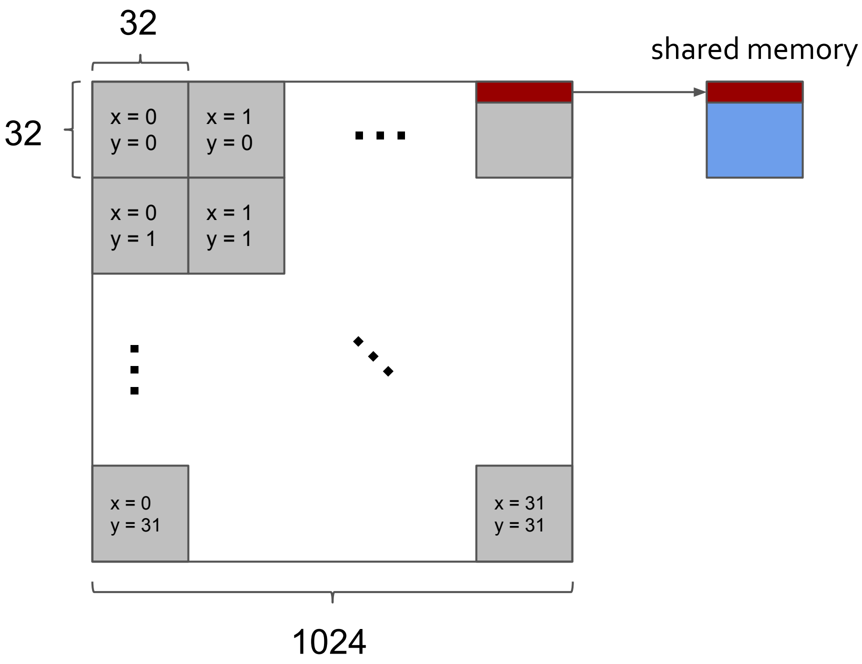
What happens if we use shared memory to do the same copy?

```
__global__ void copySharedMem(float *odata, const float *idata)
{
    __shared__ float tile[TILE_SIZE * TILE_SIZE];

    int x = blockIdx.x * TILE_SIZE + threadIdx.x;
    int y = blockIdx.y * TILE_SIZE + threadIdx.y;
    int width = gridDim.x * TILE_SIZE;

    for (int j = 0; j < TILE_SIZE; j += BLOCK_SIZE_Y) {
        tile[(threadIdx.y + j) * TILE_SIZE + threadIdx.x] =
            idata[(y + j) * width + x];
    }
    __syncthreads();

    for (int j = 0; j < TILE_SIZE; j += BLOCK_SIZE_Y) {
        odata[(y + j) * width + x] =
            tile[(threadIdx.y + j) * TILE_SIZE + threadIdx.x];
    }
}
```

Is __syncthreads() necessary?

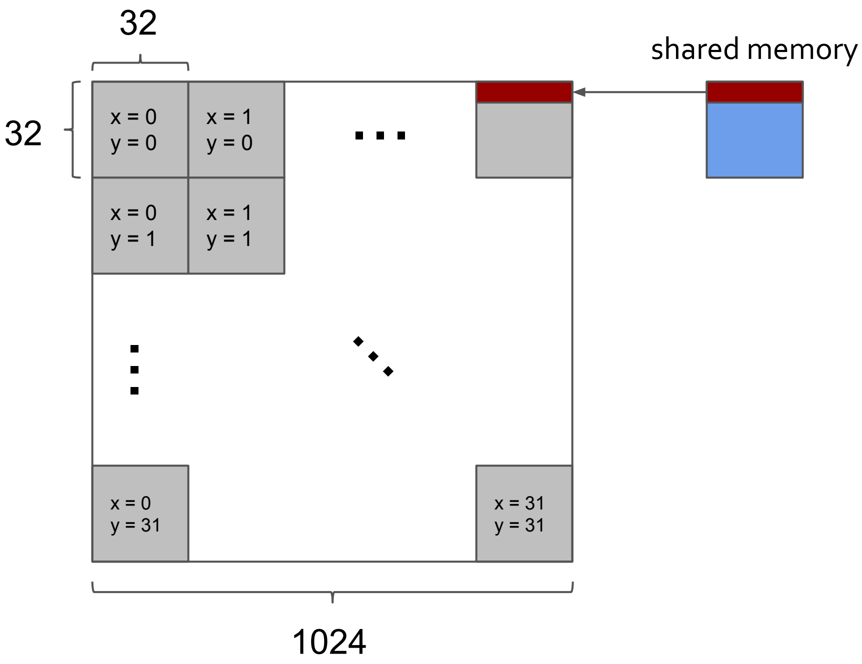
Shared Memory

```
for (int j = 0; j < TILE_SIZE; j += BLOCK_SIZE_Y) {
    tile[(threadIdx.y + j) * TILE_SIZE + threadIdx.x] =
        idata[(y + j) * width + x];
}
__syncthreads();
```



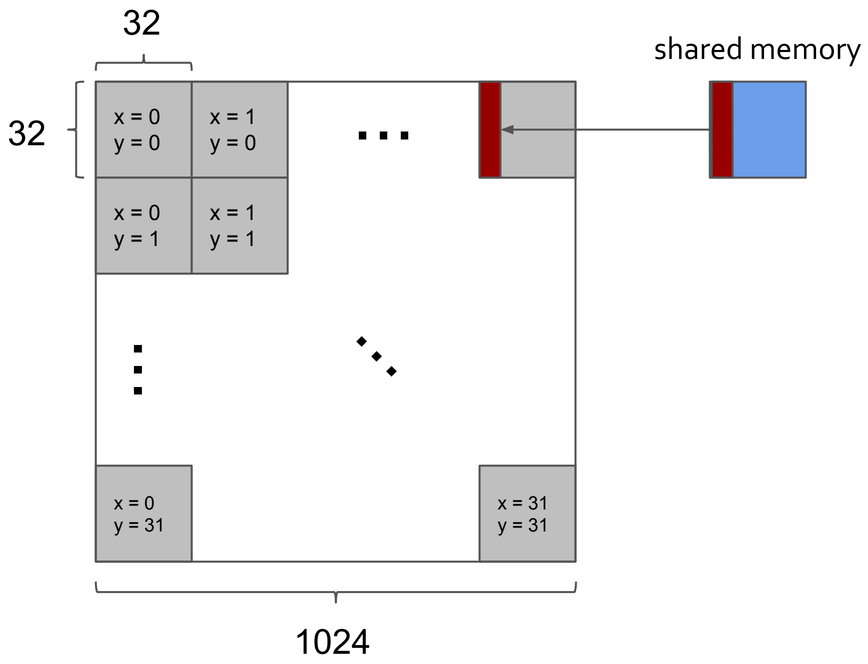
Shared Memory

```
for (int j = 0; j < TILE_SIZE; j += BLOCK_SIZE_Y) {
    odata[(y + j) * width + x] =
        tile[(threadIdx.y + j) * TILE_SIZE + threadIdx.x];
}
```



Shared Memory

`__syncthreads()` is necessary in this case



Shared Memory

Version	Bandwidth (GB/s)
copy	152.34
shared memory copy	147.97

Questions?

So far, nothing out of the ordinary

Shred Memory

Let's do a real transpose.

What if we simply read and write naively from the global memory?

```
__global__ void transposeNaive(float *odata, const float *idata)
{
    int x = blockIdx.x * TILE_SIZE + threadIdx.x;
    int y = blockIdx.y * TILE_SIZE + threadIdx.y;
    int width = gridDim.x * TILE_SIZE;

    for (int j = 0; j < TILE_SIZE; j+= BLOCK_SIZE_Y) {
        odata[x * width + (y + j)] = idata[(y + j) * width + x];
    }
}
```

Write to (j,i)

Read from (i,j)

Is the read/write coalesced (i.e., are the thread reading/writing consecutive piece of data from memory)?

Shred Memory

What if we simply read and write naively?

```
__global__ void transposeNaive(float *odata, const float *idata)
{
    int x = blockIdx.x * TILE_SIZE + threadIdx.x;
    int y = blockIdx.y * TILE_SIZE + threadIdx.y;
    int width = gridDim.x * TILE_SIZE;

    for (int j = 0; j < TILE_SIZE; j+= BLOCK_SIZE_Y) {
        odata[x * width + (y + j)] = idata[(y + j) * width + x];
    }
}
```

Read is coalesced - good

- Each warp is made up of threads consecutive in threadIdx.x (i.e., threadIdx.x = 0 ~ threadIdx.x = 31 belong to the same warp).
- Therefore, threads in each warp request 32 data elements consecutive in memory (i.e., coalesced).

Shred Memory

What if we simply read and write naively?

```
__global__ void transposeNaive(float *odata, const float *idata)
{
    int x = blockIdx.x * TILE_SIZE + threadIdx.x;
    int y = blockIdx.y * TILE_SIZE + threadIdx.y;
    int width = gridDim.x * TILE_SIZE;

    for (int j = 0; j < TILE_SIZE; j+= BLOCK_SIZE_Y) {
        odata[x * width + (y + j)] = idata[(y + j) * width + x];
    }
}
```

Write is NOT coalesced - bad

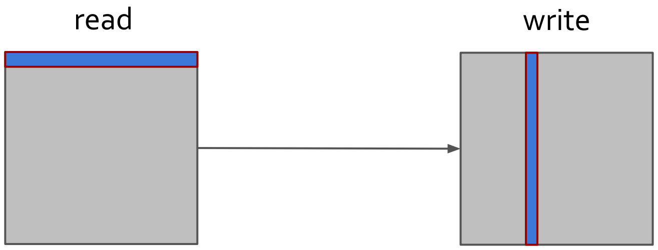
Each threads consecutive in threadIdx.x write in a column-wise manner to the memory (i.e., each write is apart by width elements apart).

Shred Memory

```
for (int j = 0; j < TILE_SIZE; j+= BLOCK_SIZE_Y) {
    odata[x * width + (y + j)] = idata[(y + j) * width + x];
}
```

Each warp reads in a coalesced manner (consecutive data in memory).

Each warp writes to location 1024 * 4 bytes apart (scatter).



Shared Memory

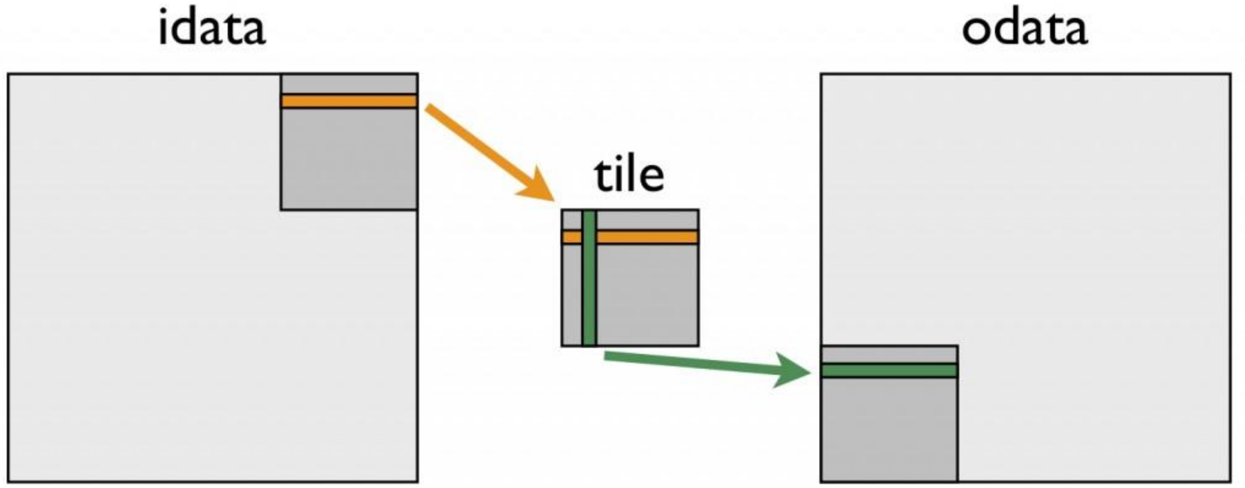
Version	Bandwidth (GB/s)
copy	152.34
shared memory copy	147.97
naive (in-memory) transpose	43.59

Because of the non-coalesced write, the performance suffers

Shared Memory

How do we make both read and write coalesced?

- Use shared memory to "rearrange" the data
- While reading from the shared memory in a non-coalesced manner is also bad, because shared memory is extremely fast, the penalty is much smaller

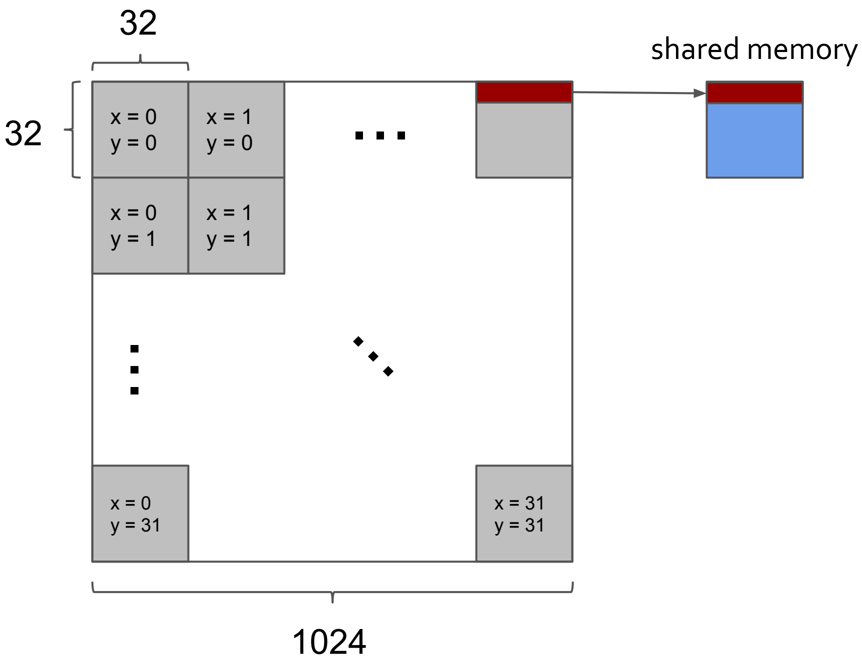


Shared Memory

```
__global__ void transposeCoalesced(float *odata, const float *idata)
{
    __shared__ float tile[TILE_SIZE][TILE_SIZE]; // tile[rows][columns]

    int x = blockIdx.x * TILE_SIZE + threadIdx.x;
    int y = blockIdx.y * TILE_SIZE + threadIdx.y;
    int width = gridDim.x * TILE_SIZE;

    for (int j = 0; j < TILE_SIZE; j += BLOCK_SIZE_Y) {
        tile[threadIdx.y + j][threadIdx.x]= idata[(y + j) * width + x];
    }
    __syncthreads();

    x = blockIdx.y * TILE_SIZE + threadIdx.x;  // transpose block offset
    y = blockIdx.x * TILE_SIZE + threadIdx.y;

  for (int j = 0; j < TILE_SIZE; j += BLOCK_SIZE_Y) {
     odata[(y + j) * width + x] = tile[threadIdx.x][threadIdx.y + j];
  }
}
```

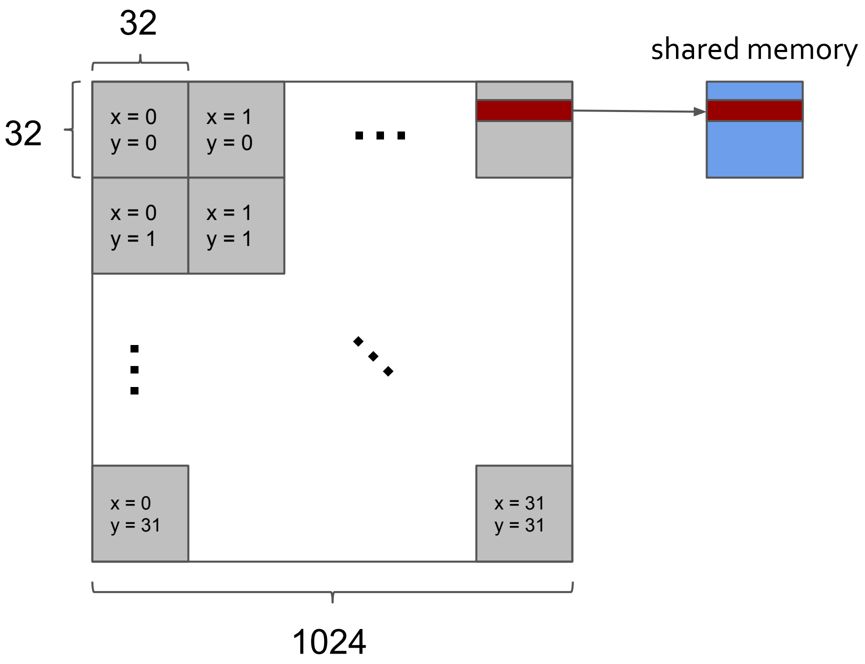
Shared Memory

```
for (int j = 0; j < TILE_SIZE; j += BLOCK_SIZE_Y) {
    tile[threadIdx.y + j][threadIdx.x] = idata[(y + j) * width + x];
}
__syncthreads();
```



Shared Memory

```
for (int j = 0; j < TILE_SIZE; j += BLOCK_SIZE_Y) {
    tile[threadIdx.y + j][threadIdx.x] = idata[(y + j) * width + x];
}
__syncthreads();
```



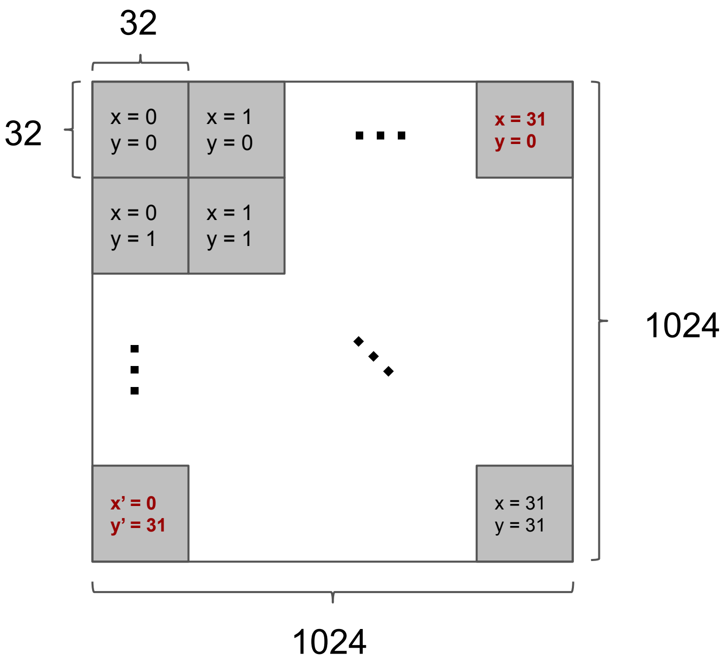
Shared Memory

```
__global__ void transposeCoalesced(float *odata, const float *idata)
{
    __shared__ float tile[TILE_SIZE][TILE_SIZE];

    int x = blockIdx.x * TILE_SIZE + threadIdx.x;
    int y = blockIdx.y * TILE_SIZE + threadIdx.y;
    int width = gridDim.x * TILE_SIZE;

    for (int j = 0; j < TILE_SIZE; j += BLOCK_SIZE_Y) {
        tile[threadIdx.y + j][threadIdx.x] = idata[(y + j) * width + x];
    }
    __syncthreads();

    x = blockIdx.y * TILE_SIZE + threadIdx.x;  // transpose block offset
    y = blockIdx.x * TILE_SIZE + threadIdx.y;

  for (int j = 0; j < TILE_SIZE; j += BLOCK_SIZE_Y) {
     odata[(y + j) * width + x] = tile[threadIdx.x][threadIdx.y + j];
  }
}
```

Recalculate index for write

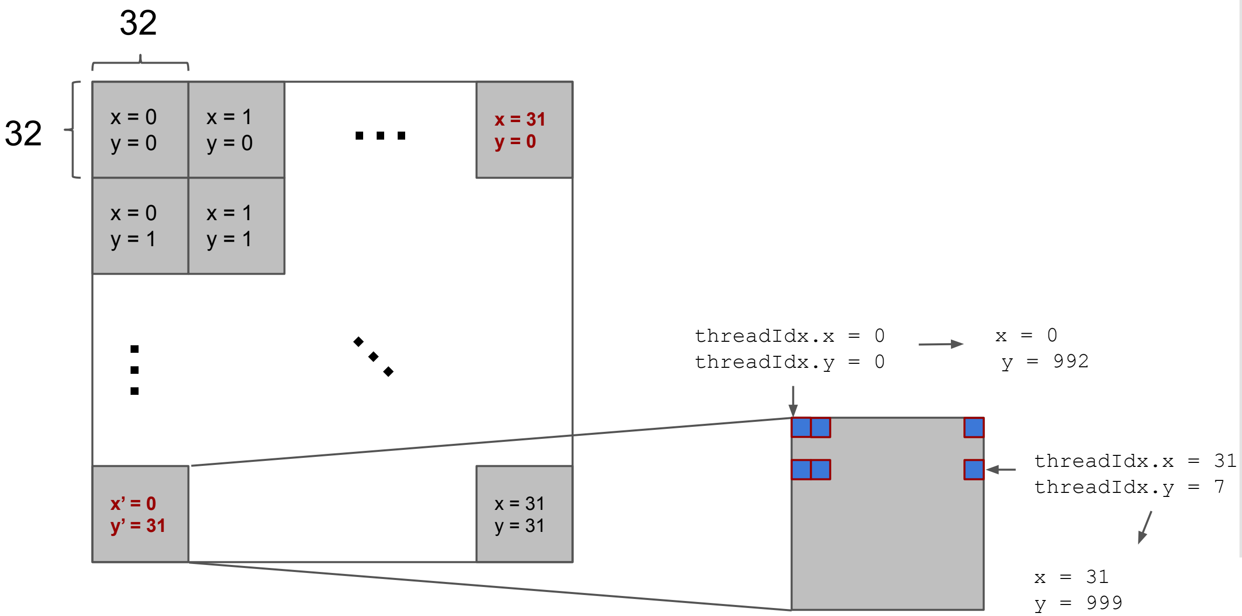
Shared Memory

```
x = blockIdx.y * TILE_SIZE + threadIdx.x;  // transpose block offset
y = blockIdx.x * TILE_SIZE + threadIdx.y;
```



Shared Memory

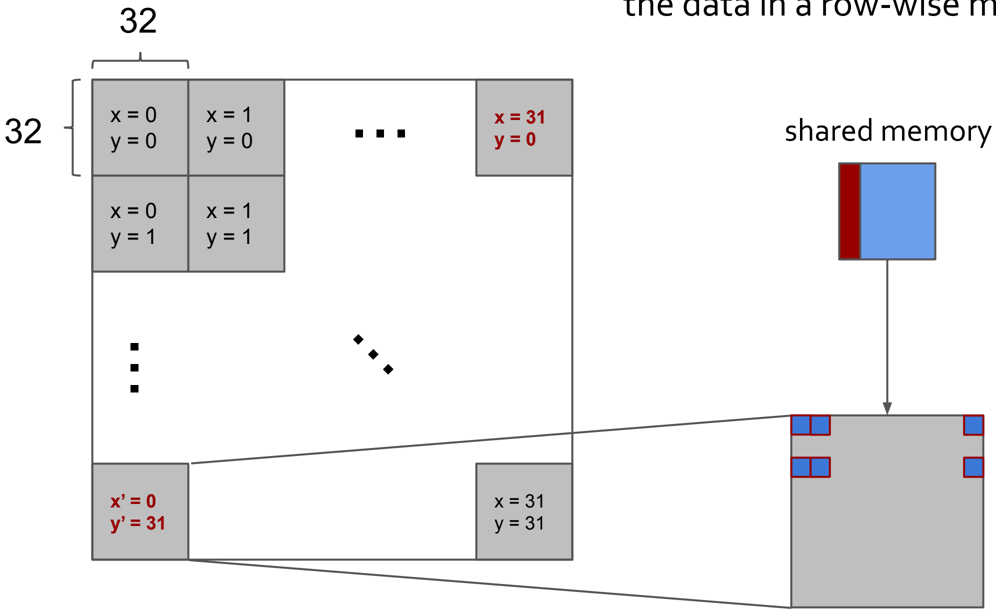
```
x = blockIdx.y * TILE_SIZE + threadIdx.x;  // transpose block offset
y = blockIdx.x * TILE_SIZE + threadIdx.y;
```



Shared Memory

```
for (int j = 0; j < TILE_SIZE; j += BLOCK_SIZE_Y) {
    odata[(y + j) * width + x] = tile[threadIdx.x][threadIdx.y + j];
}
```

Each thread reads from a column in the shared memory, then write the data in a row-wise manner



Shared Memory

Version	Bandwidth (GB/s)
copy	152.34
shared memory copy	147.97
naive (in-memory) transpose	43.59
coalesced transpose	101.76

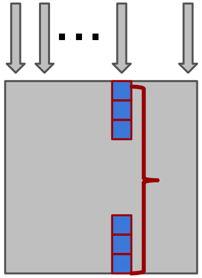
Shared Memory

We are getting 101.76 GB/s (out of 152.34 possible)?

Shared memory has 32 **banks**

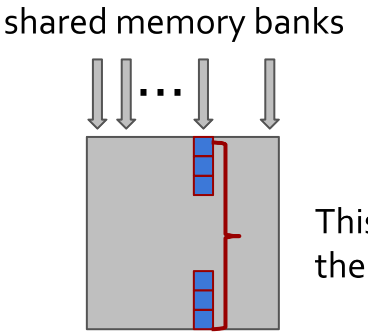
- Banks are basically like doors where data can come out of - more ports mean higher bandwidth
- Access that are 32 elements apart are read from the same bank
- Reading from shared memory column-wise reads from the same bank - reads are serialized

shared memory banks



This column of data are accessed from the same bank

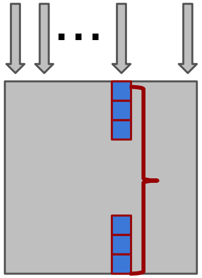
Shared Memory



This column of data are accessed from the same bank

Shared Memory

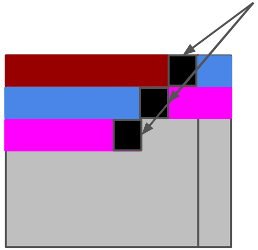
shared memory banks



This column of data are accessed from the same bank

```
__shared__ float tile[TILE_SIZE][TILE_SIZE + 1];
```

Padding element



By inserting a "padding" element, each elements are shifted by one, and columns are no longer read from the same bank

Shared Memory

Version	Bandwidth (GB/s)
copy	152.34
shared memory copy	147.97
naive (in-memory) transpose	43.59
coalesced transpose	101.76
(bank) conflict-free transpose	134.70

Shared Memory

Volta V100 GPU, 900 GB/s peak memory bandwidth

Version	Bandwidth (GB/s)
copy	766.59
shared memory copy	782.32
naive (in-memory) transpose	208.81
coalesced transpose	648.26
(bank) conflict-free transpose	717.38

Questions?

Typically, shared memory acts as cache - keeping the data and reusing the data reduces data access latency (from DRAM) and improves performance

However, that is not the only way of using shared memory - by using the (extremely fast) shared memory as an intermediate buffer/cache, we can minimize the penalty of non-coalesced access

- Note that with the transpose example, data was NOT reused at all